 FreeRTOS内核是高度可定制的，使用配置文件FreeRTOSConfig.h进行定制。每个FreeRTOS应用都必须包含这个头文件，用户根据实际应用来裁剪定制FreeRTOS内核。这个配置文件是针对用户程序的，而非内核，因此配置文件一般放在应用程序目录下，不要放在RTOS内核源码目录下。

       在下载的FreeRTOS文件包中，每个演示例程都有一个FreeRTOSConfig.h文件。有些例程的配置文件是比较旧的版本，可能不会包含所有有效选项。如果没有在配置文件中指定某个选项，那么RTOS内核会使用默认值。典型的FreeRTOSConfig.h配置文件定义如下所示，随后会说明里面的每一个参数。

#ifndef FREERTOS\_CONFIG\_H

#define FREERTOS\_CONFIG\_H

/\*Here is a good place to include header files that are required across

yourapplication. \*/

#include "something.h"

#define configUSE\_PREEMPTION 1

#define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 0

#define configUSE\_TICKLESS\_IDLE 0

#define configCPU\_CLOCK\_HZ 60000000

#define configTICK\_RATE\_HZ 250

#define configMAX\_PRIORITIES 5

#define configMINIMAL\_STACK\_SIZE 128

#define configTOTAL\_HEAP\_SIZE 10240

#define configMAX\_TASK\_NAME\_LEN 16

#define configUSE\_16\_BIT\_TICKS 0

#define configIDLE\_SHOULD\_YIELD 1

#define configUSE\_TASK\_NOTIFICATIONS 1

#define configUSE\_MUTEXES 0

#define configUSE\_RECURSIVE\_MUTEXES 0

#define configUSE\_COUNTING\_SEMAPHORES 0

#define configUSE\_ALTERNATIVE\_API 0/\* Deprecated! \*/

#define configQUEUE\_REGISTRY\_SIZE 10

#define configUSE\_QUEUE\_SETS 0

#define configUSE\_TIME\_SLICING 0

#define configUSE\_NEWLIB\_REENTRANT 0

#define configENABLE\_BACKWARD\_COMPATIBILITY 0

#define configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS 5

/\*Hook function related definitions. \*/

#define configUSE\_IDLE\_HOOK 0

#define configUSE\_TICK\_HOOK 0

#define configCHECK\_FOR\_STACK\_OVERFLOW 0

#define configUSE\_MALLOC\_FAILED\_HOOK 0

/\*Run time and task stats gathering related definitions. \*/

#define configGENERATE\_RUN\_TIME\_STATS 0

#define configUSE\_TRACE\_FACILITY 0

#define configUSE\_STATS\_FORMATTING\_FUNCTIONS 0

/\*Co-routine related definitions. \*/

#define configUSE\_CO\_ROUTINES 0

#define configMAX\_CO\_ROUTINE\_PRIORITIES 1

/\*Software timer related definitions. \*/

#define configUSE\_TIMERS 1

#define configTIMER\_TASK\_PRIORITY 3

#define configTIMER\_QUEUE\_LENGTH 10

#define configTIMER\_TASK\_STACK\_DEPTH configMINIMAL\_STACK\_SIZE

/\*Interrupt nesting behaviour configuration. \*/

#define configKERNEL\_INTERRUPT\_PRIORITY [dependent of processor]

#define configMAX\_SYSCALL\_INTERRUPT\_PRIORITY [dependent on processor and application]

#define configMAX\_API\_CALL\_INTERRUPT\_PRIORITY [dependent on processor and application]

/\*Define to trap errors during development. \*/

#define configASSERT( ( x ) ) if( ( x ) == 0) vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

/\*FreeRTOS MPU specific definitions. \*/

#define configINCLUDE\_APPLICATION\_DEFINED\_PRIVILEGED\_FUNCTIONS 0

/\*Optional functions - most linkers will remove unused functions anyway. \*/

#define INCLUDE\_vTaskPrioritySet 1

#define INCLUDE\_uxTaskPriorityGet 1

#define INCLUDE\_vTaskDelete 1

#define INCLUDE\_vTaskSuspend 1

#define INCLUDE\_xResumeFromISR 1

#define INCLUDE\_vTaskDelayUntil 1

#define INCLUDE\_vTaskDelay 1

#define INCLUDE\_xTaskGetSchedulerState 1

#define INCLUDE\_xTaskGetCurrentTaskHandle 1

#define INCLUDE\_uxTaskGetStackHighWaterMark 0

#define INCLUDE\_xTaskGetIdleTaskHandle 0

#define INCLUDE\_xTimerGetTimerDaemonTaskHandle 0

#define INCLUDE\_pcTaskGetTaskName 0

#define INCLUDE\_eTaskGetState 0

#define INCLUDE\_xEventGroupSetBitFromISR 1

#define INCLUDE\_xTimerPendFunctionCall 0

/\* Aheader file that defines trace macro can be included here. \*/

#end if/\* FREERTOS\_CONFIG\_H\*/

**1.configUSE\_PREEMPTION:**

       为1时RTOS使用抢占式调度器，为0时RTOS使用协作式调度器（时间片）。

      注：在多任务管理机制上，操作系统可以分为抢占式和协作式两种。协作式操作系统是任务主动释放CPU后，切换到下一个任务。任务切换的时机完全取决于正在运行的任务。

**2.configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION:**

       某些运行FreeRTOS的硬件有两种方法选择下一个要执行的任务：通用方法和特定于硬件的方法（以下简称“特殊方法”）。

**通用方法：**

configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为0或者硬件不支持这种特殊方法。

可以用于所有FreeRTOS支持的硬件。

完全用C实现，效率略低于特殊方法。

不强制要求限制最大可用优先级数目

**特殊方法：**

并非所有硬件都支持。

必须将configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION设置为1。

依赖一个或多个特定架构的汇编指令（一般是类似计算前导零[CLZ]指令）。

比通用方法更高效。

一般强制限定最大可用优先级数目为32。

**3.configUSE\_TICKLESS\_IDLE:**

       设置configUSE\_TICKLESS\_IDLE为1使能低功耗tickless模式，为0保持系统节拍（tick）中断一直运行。

       通常情况下，FreeRTOS回调空闲任务钩子函数（需要设计者自己实现），在空闲任务钩子函数中设置微处理器进入低功耗模式来达到省电的目的。因为系统要响应系统节拍中断事件，因此使用这种方法会周期性的退出、再进入低功耗状态。如果系统节拍中断频率过快，则大部分电能和CPU时间会消耗在进入和退出低功耗状态上。

       FreeRTOS的tickless空闲模式会在空闲周期时停止周期性系统节拍中断。停止周期性系统节拍中断可以使微控制器长时间处于低功耗模式。移植层需要配置外部唤醒中断，当唤醒事件到来时，将微控制器从低功耗模式唤醒。微控制器唤醒后，会重新使能系统节拍中断。由于微控制器在进入低功耗后，系统节拍计数器是停止的，但我们又需要知道这段时间能折算成多少次系统节拍中断周期，这就需要有一个不受低功耗影响的外部时钟源，即微处理器处于低功耗模式时它也在计时的，这样在重启系统节拍中断时就可以根据这个外部计时器计算出一个调整值并写入RTOS 系统节拍计数器变量中。

**4.configUSE\_IDLE\_HOOK:**

       设置为1使用空闲钩子（Idle Hook类似于回调函数），0忽略空闲钩子。

       当RTOS调度器开始工作后，为了保证至少有一个任务在运行，空闲任务被自动创建，占用最低优先级（0优先级）。对于已经删除的RTOS任务，空闲任务可以释放分配给它们的堆栈内存。因此，在应用中应该注意，使用vTaskDelete()函数时要确保空闲任务获得一定的处理器时间。除此之外，空闲任务没有其它特殊功能，因此可以任意的剥夺空闲任务的处理器时间。

       应用程序也可能和空闲任务共享同个优先级。

       空闲任务钩子是一个函数，这个函数由用户来实现，RTOS规定了函数的名字和参数，这个函数在每个空闲任务周期都会被调用。

       要创建一个空闲钩子：

 设置FreeRTOSConfig.h 文件中的configUSE\_IDLE\_HOOK 为1；

 定义一个函数，函数名和参数如下所示：

void vApplicationIdleHook(void );

       这个钩子函数不可以调用会引起空闲任务阻塞的API函数（例如：vTaskDelay()、带有阻塞时间的队列和信号量函数），在钩子函数内部使用协程是被允许的。

       使用空闲钩子函数设置CPU进入省电模式是很常见的。

**5.configUSE\_MALLOC\_FAILED\_HOOK:**

       每当一个任务、队列、信号量被创建时，内核使用一个名为pvPortMalloc()的函数来从堆中分配内存。官方的下载包中包含5个简单内存分配策略，分别保存在源文件heap\_1.c、heap\_2.c、heap\_3.c、heap\_4.c、heap\_5.c中。        仅当使用这五个简单策略之一时，宏configUSE\_MALLOC\_FAILED\_HOOK才有意义。

       如果定义并正确配置malloc()失败钩子函数，则这个函数会在pvPortMalloc()函数返回NULL时被调用。只有FreeRTOS在响应内存分配请求时发现堆内存不足才会返回NULL。

       如果宏configUSE\_MALLOC\_FAILED\_HOOK设置为1，那么必须定义一个malloc()失败钩子函数，如果宏configUSE\_MALLOC\_FAILED\_HOOK设置为0，malloc()失败钩子函数不会被调用，即便已经定义了这个函数。malloc()失败钩子函数的函数名和原型必须如下所示：

void vApplicationMallocFailedHook( void);

**6.configUSE\_TICK\_HOOK:**

       设置为1使用时间片钩子（Tick Hook），0忽略时间片钩子。

       注：时间片钩子函数（Tick Hook Function）

       时间片中断可以周期性的调用一个被称为钩子函数（回调函数）的应用程序。时间片钩子函数可以很方便的实现一个定时器功能。

       只有在FreeRTOSConfig.h中的configUSE\_TICK\_HOOK设置成1时才可以使用时间片钩子。一旦此值设置成1，就要定义钩子函数，函数名和参数如下所示：

void vApplicationTickHook( void );

       vApplicationTickHook()函数在中断服务程序中执行，因此这个函数必须非常短小，不能大量使用堆栈，不能调用任何不是以”FromISR" 或 "FROM\_ISR”结尾的API函数。

       在FreeRTOSVx.x.x\FreeRTOS\Demo\Common\Minimal文件夹下的crhook.c文件中有使用时间片钩子函数的例程。

**7.configCPU\_CLOCK\_HZ:**

       写入实际的CPU内核时钟频率，也就是CPU指令执行频率，通常称为Fcclk。配置此值是为了正确的配置系统节拍中断周期。

**8.configTICK\_RATE\_HZ:**

       RTOS 系统节拍中断的频率。即一秒中断的次数，每次中断RTOS都会进行任务调度。

系统节拍中断用来测量时间，因此，越高的测量频率意味着可测到越高的分辨率时间。但是，高的系统节拍中断频率也意味着RTOS内核占用更多的CPU时间，因此会降低效率。RTOS演示例程都是使用系统节拍中断频率为1000HZ，这是为了测试RTOS内核，比实际使用的要高。（实际使用时不用这么高的系统节拍中断频率）

       多个任务可以共享一个优先级，RTOS调度器为相同优先级的任务分享CPU时间，在每一个RTOS 系统节拍中断到来时进行任务切换。高的系统节拍中断频率会降低分配给每一个任务的“时间片”持续时间。

**9.configMAX\_PRIORITIES:**

       配置应用程序有效的优先级数目。任何数量的任务都可以共享一个优先级，使用协程可以单独的给与它们优先权。见configMAX\_CO\_ROUTINE\_PRIORITIES。

       在RTOS内核中，每个有效优先级都会消耗一定量的RAM，因此这个值不要超过你的应用实际需要的优先级数目。

注：任务优先级

       每一个任务都会被分配一个优先级，优先级值从0~ （configMAX\_PRIORITIES - 1）之间。低优先级数表示低优先级任务。空闲任务的优先级为0（tskIDLE\_PRIORITY），因此它是最低优先级任务。

       FreeRTOS调度器将确保处于就绪状态（Ready）或运行状态（Running）的高优先级任务比同样处于就绪状态的低优先级任务优先获取处理器时间。换句话说，处于运行状态的任务永远是高优先级任务。

       处于就绪状态的相同优先级任务使用时间片调度机制共享处理器时间。

**10.configMINIMAL\_STACK\_SIZE:**

       定义空闲任务使用的堆栈大小。通常此值不应小于对应处理器演示例程文件FreeRTOSConfig.h中定义的数值。

       就像xTaskCreate()函数的堆栈大小参数一样，堆栈大小不是以字节为单位而是以字为单位的，比如在32位架构下，栈大小为100表示栈内存占用400字节的空间。

**11.configTOTAL\_HEAP\_SIZE:**

       RTOS内核总计可用的有效的RAM大小。仅在你使用官方下载包中附带的内存分配策略时，才有可能用到此值。每当创建任务、队列、互斥量、软件定时器或信号量时，RTOS内核会为此分配RAM，这里的RAM都属于configTOTAL\_HEAP\_SIZE指定的内存区。后续的内存配置会详细讲到官方给出的内存分配策略。

**12.configMAX\_TASK\_NAME\_LEN:**

       调用任务函数时，需要设置描述任务信息的字符串，这个宏用来定义该字符串的最大长度。这里定义的长度包括字符串结束符’\0’。

**13.configUSE\_TRACE\_FACILITY:**

       设置成1表示启动可视化跟踪调试，会激活一些附加的结构体成员和函数。

**14.configUSE\_STATS\_FORMATTING\_FUNCTIONS （V7.5.0新增）：**

       设置宏configUSE\_TRACE\_FACILITY和configUSE\_STATS\_FORMATTING\_FUNCTIONS为1会编译vTaskList()和vTaskGetRunTimeStats()函数。如果将这两个宏任意一个设置为0，上述两个函数不会被编译。

**15.configUSE\_16\_BIT\_TICKS：**

       定义系统节拍计数器的变量类型，即定义portTickType是表示16位变量还是32位变量。

       定义configUSE\_16\_BIT\_TICKS为1意味着portTickType代表16位无符号整形，定义configUSE\_16\_BIT\_TICKS为0意味着portTickType代表32位无符号整形。

       使用16位类型可以大大提高8位和16位架构微处理器的性能，但这也限制了最大时钟计数为65535个’Tick’。因此，如果Tick频率为250HZ（4MS中断一次），对于任务最大延时或阻塞时间，16位计数器是262秒，而32位是17179869秒。

**16.configIDLE\_SHOULD\_YIELD：**

       这个参数控制任务在空闲优先级中的行为。仅在满足下列条件后，才会起作用。

使用抢占式内核调度

用户任务使用空闲优先级。

         通过时间片共享同一个优先级的多个任务，如果共享的优先级大于空闲优先级，并假设没有更高优先级任务，这些任务应该获得相同的处理器时间。

         但如果共享空闲优先级时，情况会稍微有些不同。当configIDLE\_SHOULD\_YIELD为1时，其它共享空闲优先级的用户任务就绪时，空闲任务立刻让出CPU，用户任务运行，这样确保了能最快响应用户任务。处于这种模式下也会有不良效果（取决于你的程序需要），描述如下：

       图中描述了四个处于空闲优先级的任务，任务A、B和C是用户任务，任务I是空闲任务。上下文切换周期性的发生在T0、T1…T6时刻。当用户任务运行时，空闲任务立刻让出CPU，但是，空闲任务已经消耗了当前时间片中的一定时间。这样的结果就是空闲任务I和用户任务A共享一个时间片。用户任务B和用户任务C因此获得了比用户任务A更多的处理器时间。

       可以通过下面方法避免：

如果合适的话，将处于空闲优先级的各单独的任务放置到空闲钩子函数中；

创建的用户任务优先级大于空闲优先级；

设置IDLE\_SHOULD\_YIELD为0；

       设置configIDLE\_SHOULD\_YIELD为0将阻止空闲任务为用户任务让出CPU，直到空闲任务的时间片结束。这确保所有处在空闲优先级的任务分配到相同多的处理器时间，但是，这是以分配给空闲任务更高比例的处理器时间为代价的。

**17.configUSE\_TASK\_NOTIFICATIONS（V8.2.0新增）：**

       设置宏configUSE\_TASK\_NOTIFICATIONS为1（或不定义宏configUSE\_TASK\_NOTIFICATIONS）将会开启任务通知功能，有关的API函数也会被编译。设置宏configUSE\_TASK\_NOTIFICATIONS为0则关闭任务通知功能，相关API函数也不会被编译。默认这个功能是开启的。开启后，每个任务多增加8字节RAM。

       这是个很有用的特性，一大亮点。

       每个RTOS任务具有一个32位的通知值，RTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除任务的阻塞状态（因等待任务通知而进入阻塞状态）。相对于以前必须分别创建队列、二进制信号量、计数信号量或事件组的情况，使用任务通知显然更灵活。更好的是，相比于使用信号量解除任务阻塞，使用任务通知可以快45%（使用GCC编译器，-o2优化级别）。

**18.configUSE\_MUTEXES：**

       设置为1表示使用互斥量，设置成0表示忽略互斥量。读者应该了解在FreeRTOS中互斥量和二进制信号量的区别。

       关于互斥量和二进制信号量简单说：

互斥型信号量必须是同一个任务申请，同一个任务释放，其他任务释放无效。

二进制信号量，一个任务申请成功后，可以由另一个任务释放。

互斥型信号量是二进制信号量的子集

**19.configUSE\_RECURSIVE\_MUTEXES：**

       设置成1表示使用递归互斥量，设置成0表示不使用。

**20.configUSE\_COUNTING\_SEMAPHORES：**

       设置成1表示使用计数信号量，设置成0表示不使用。

**21.configUSE\_ALTERNATIVE\_API：**

       设置成1表示使用“替代”队列函数（'alternative' queue functions），设置成0不使用。替代API在queue.h头文件中有详细描述。

        注：“替代”队列函数已经被弃用，在新的设计中不要使用它！

**22.configCHECK\_FOR\_STACK\_OVERFLOW：**

       每个任务维护自己的栈空间，任务创建时会自动分配任务需要的占内存，分配内存大小由创建任务函数（xTaskCreate()）的一个参数指定。堆栈溢出是设备运行不稳定的最常见原因，因此FreeeRTOS提供了两个可选机制用来辅助检测和改正堆栈溢出。配置宏configCHECK\_FOR\_STACK\_OVERFLOW为不同的常量来使用不同堆栈溢出检测机制。

       注意，这个选项仅适用于内存映射未分段的微处理器架构。并且，在RTOS检测到堆栈溢出发生之前，一些处理器可能先产生故障（fault）或异常（exception）来反映堆栈使用的恶化。如果宏configCHECK\_FOR\_STACK\_OVERFLOW没有设置成0，用户必须提供一个栈溢出钩子函数，这个钩子函数的函数名和参数必须如下所示：

void vApplicationStackOverflowHook(TaskHandle\_t xTask, signed char \*pcTaskName );

       参数xTask和pcTaskName为堆栈溢出任务的句柄和名字。请注意，如果溢出非常严重，这两个参数信息也可能是错误的！在这种情况下，可以直接检查pxCurrentTCb变量。

       推荐仅在开发或测试阶段使用栈溢出检查，因为堆栈溢出检测会增大上下文切换开销。

       任务切换出去后，该任务的上下文环境被保存到自己的堆栈空间，这时很可能堆栈的使用量达到了最大（最深）值。在这个时候，RTOS内核会检测堆栈指针是否还指向有效的堆栈空间。如果堆栈指针指向了有效堆栈空间之外的地方，堆栈溢出钩子函数会被调用。

       这个方法速度很快，但是不能检测到所有堆栈溢出情况（比如，堆栈溢出没有发生在上下文切换时）。设置configCHECK\_FOR\_STACK\_OVERFLOW为1会使用这种方法。

       当堆栈首次创建时，在它的堆栈区中填充一些已知值（标记）。当任务切换时，RTOS内核会检测堆栈最后的16个字节，确保标记数据没有被覆盖。如果这16个字节有任何一个被改变，则调用堆栈溢出钩子函数。

       这个方法比第一种方法要慢，但也相当快了。它能有效捕捉堆栈溢出事件（即使堆栈溢出没有发生在上下文切换时），但是理论上它也不能百分百的捕捉到所有堆栈溢出（比如堆栈溢出的值和标记值相同，当然，这种情况发生的概率极小）。

       使用这个方法需要设置configCHECK\_FOR\_STACK\_OVERFLOW为2.

**23.configQUEUE\_REGISTRY\_SIZE：**

       队列记录有两个目的，都涉及到RTOS内核的调试：

它允许在调试GUI中使用一个队列的文本名称来简单识别队列；

包含调试器需要的每一个记录队列和信号量定位信息；

       除了进行内核调试外，队列记录没有其它任何目的。

       configQUEUE\_REGISTRY\_SIZE定义可以记录的队列和信号量的最大数目。如果你想使用RTOS内核调试器查看队列和信号量信息，则必须先将这些队列和信号量进行注册，只有注册后的队列和信号量才可以使用RTOS内核调试器查看。查看API参考手册中的vQueueAddToRegistry() 和vQueueUnregisterQueue()函数获取更多信息。

**24.configUSE\_QUEUE\_SETS：**

       设置成1使能队列集功能（可以阻塞、挂起到多个队列和信号量），设置成0取消队列集功能。

**25.configUSE\_TIME\_SLICING（V7.5.0新增）:**

       默认情况下（宏configUSE\_TIME\_SLICING未定义或者宏configUSE\_TIME\_SLICING设置为1），FreeRTOS使用基于时间片的优先级抢占式调度器。这意味着RTOS调度器总是运行处于最高优先级的就绪任务，在每个RTOS 系统节拍中断时在相同优先级的多个任务间进行任务切换。如果宏configUSE\_TIME\_SLICING设置为0，RTOS调度器仍然总是运行处于最高优先级的就绪任务，但是当RTOS 系统节拍中断发生时，相同优先级的多个任务之间不再进行任务切换。

**26.configUSE\_NEWLIB\_REENTRANT（V7.5.0新增）:**

       如果宏configUSE\_NEWLIB\_REENTRANT设置为1，每一个创建的任务会分配一个newlib（一个嵌入式C库）reent结构。

**27.configENABLE\_BACKWARD\_COMPATIBILITY:**

       头文件FreeRTOS.h包含一系列#define宏定义，用来映射版本V8.0.0和V8.0.0之前版本的数据类型名字。这些宏可以确保RTOS内核升级到V8.0.0或以上版本时，之前的应用代码不用做任何修改。在FreeRTOSConfig.h文件中设置宏configENABLE\_BACKWARD\_COMPATIBILITY为0会去掉这些宏定义，并且需要用户确认升级之前的应用没有用到这些名字。

**28.configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS:**

       设置每个任务的线程本地存储指针数组大小。

       线程本地存储允许应用程序在任务的控制块中存储一些值，每个任务都有自己独立的储存空间，宏configNUM\_THREAD\_LOCAL\_STORAGE\_POINTERS指定每个任务线程本地存储指针数组的大小。API函数vTaskSetThreadLocalStoragePointer()用于向指针数组中写入值，API函数pvTaskGetThreadLocalStoragePointer()用于从指针数组中读取值。

       比如，许多库函数都包含一个叫做errno的全局变量。某些库函数使用errno返回库函数错误信息，应用程序检查这个全局变量来确定发生了那些错误。在单线程程序中，将errno定义成全局变量是可以的，但是在多线程应用中，每个线程（任务）必须具有自己独有的errno值，否则，一个任务可能会读取到另一个任务的errno值。

       FreeRTOS提供了一个灵活的机制，使得应用程序可以使用线程本地存储指针来读写线程本地存储。具体参见后续文章《FreeRTOS系列第12篇---FreeRTOS任务应用函数》。

**29.configGENERATE\_RUN\_TIME\_STATS:**

       设置宏configGENERATE\_RUN\_TIME\_STATS为1使能运行时间统计功能。一旦设置为1，则下面两个宏必须被定义：

portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()：用户程序需要提供一个基准时钟函数，函数完成初始化基准时钟功能，这个函数要被define到宏portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS()上。这是因为运行时间统计需要一个比系统节拍中断频率还要高分辨率的基准定时器，否则，统计可能不精确。基准定时器中断频率要比统节拍中断快10~100倍。基准定时器中断频率越快，统计越精准，但能统计的运行时间也越短（比如，基准定时器10ms中断一次，8位无符号整形变量可以计到2.55秒，但如果是1秒中断一次，8位无符号整形变量可以统计到255秒）。

portGET\_RUN\_TIME\_COUNTER\_VALUE()：用户程序需要提供一个返回基准时钟当前“时间”的函数，这个函数要被define到宏portGET\_RUN\_TIME\_COUNTER\_VALUE()上。

       举一个例子，假如我们配置了一个定时器，每500us中断一次。在定时器中断服务例程中简单的使长整形变量ulHighFrequencyTimerTicks自增。那么上面提到两个宏定义如下（可以在FreeRTOSConfig.h中添加）：

extern volatile unsigned longulHighFrequencyTimerTicks;

#define portCONFIGURE\_TIMER\_FOR\_RUN\_TIME\_STATS() ( ulHighFrequencyTimerTicks = 0UL )

#define portGET\_RUN\_TIME\_COUNTER\_VALUE() ulHighFrequencyTimerTicks

**30.configUSE\_CO\_ROUTINES:**

       设置成1表示使用协程，0表示不使用协程。如果使用协程，必须在工程中包含croutine.c文件。

       注：协程（Co-routines）主要用于资源发非常受限的嵌入式系统（RAM非常少），通常不会用于32位微处理器。

       在当前嵌入式硬件环境下，不建议使用协程，FreeRTOS的开发者早已经停止开发协程。

**31.configMAX\_CO\_ROUTINE\_PRIORITIES:**

       应用程序协程（Co-routines）的有效优先级数目，任何数目的协程都可以共享一个优先级。使用协程可以单独的分配给任务优先级。见configMAX\_PRIORITIES。

**32.configUSE\_TIMERS:**

       设置成1使用软件定时器，为0不使用软件定时器功能。详细描述见FreeRTOS software timers 。

**33.configTIMER\_TASK\_PRIORITY:**

       设置软件定时器服务/守护进程的优先级。详细描述见FreeRTOS software timers 。

**34.configTIMER\_QUEUE\_LENGTH:**

       设置软件定时器命令队列的长度。详细描述见FreeRTOS software timers。

**35.configTIMER\_TASK\_STACK\_DEPTH:**

       设置软件定时器服务/守护进程任务的堆栈深度，详细描述见FreeRTOS software timers 。

**36.configKERNEL\_INTERRUPT\_PRIORITY、configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和configMAX\_API\_CALL\_INTERRUPT\_PRIORITY:**

       这是移植和应用FreeRTOS出错最多的地方，所以需要打起精神仔细读懂。

       Cortex-M3、PIC24、dsPIC、PIC32、SuperH和RX600硬件设备需要设置宏configKERNEL\_INTERRUPT\_PRIORITY；PIC32、RX600和Cortex-M硬件设备需要设置宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY。

       configMAX\_SYSCALL\_INTERRUPT\_PRIORITY和configMAX\_API\_CALL\_INTERRUPT\_PRIORITY，这两个宏是等价的，后者是前者的新名字，用于更新的移植层代码。

       注意下面的描述中，在中断服务例程中仅可以调用以“FromISR”结尾的API函数。

仅需要设置configKERNEL\_INTERRUPT\_PRIORITY的硬件设备（也就是宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY不会用到）：configKERNEL\_INTERRUPT\_PRIORITY用来设置RTOS内核自己的中断优先级。调用API函数的中断必须运行在这个优先级；不调用API函数的中断，可以运行在更高的优先级，所以这些中断不会被因RTOS内核活动而延时。

configKERNEL\_INTERRUPT\_PRIORITY和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY都需要设置的硬件设备：configKERNEL\_INTERRUPT\_PRIORITY用来设置RTOS内核自己的中断优先级。因为RTOS内核中断不允许抢占用户使用的中断，因此这个宏一般定义为硬件最低优先级。configMAX\_SYSCALL\_INTERRUPT\_PRIORITY用来设置可以在中断服务程序中安全调用FreeRTOS API函数的最高中断优先级。优先级小于等于这个宏所代表的优先级时，程序可以在中断服务程序中安全的调用FreeRTOS API函数；如果优先级大于这个宏所代表的优先级，表示FreeRTOS无法禁止这个中断，在这个中断服务程序中绝不可以调用任何API函数。

       通过设置configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的优先级级别高于configKERNEL\_INTERRUPT\_PRIORITY可以实现完整的中断嵌套模式。这意味着FreeRTOS内核不能完全禁止中断，即使在临界区。此外，这对于分段内核架构的微处理器是有利的。请注意，当一个新中断发生后，某些微处理器架构会（在硬件上）禁止中断，这意味着从硬件响应中断到FreeRTOS重新使能中断之间的这段短时间内，中断是不可避免的被禁止的。

       不调用API的中断可以运行在比configMAX\_SYSCALL\_INTERRUPT\_PRIORITY高的优先级，这些级别的中断不会被FreeRTOS禁止，因此不会因为执行RTOS内核而被延时。

       例如：假如一个微控制器有8个中断优先级别：0表示最低优先级，7表示最高优先级（Cortex-M3和Cortex-M4内核优先数和优先级别正好与之相反，后续文章会专门介绍它们）。当两个配置选项分别为4和0时，下图描述了每一个优先级别可以和不可做的事件：

configMAX\_SYSCALL\_INTERRUPT\_PRIORITY=4

configKERNEL\_INTERRUPT\_PRIORITY=0

       这些配置参数允许非常灵活的中断处理：

       在系统中可以像其它任务一样为中断处理任务分配优先级。这些任务通过一个相应中断唤醒。中断服务例程（ISR）内容应尽可能的精简---仅用于更新数据然后唤醒高优先级任务。ISR退出后，直接运行被唤醒的任务，因此中断处理（根据中断获取的数据来进行的相应处理）在时间上是连续的，就像ISR在完成这些工作。这样做的好处是当中断处理任务执行时，所有中断都可以处在使能状态。

       中断、中断服务例程（ISR）和中断处理任务是三码事：当中断来临时会进入中断服务例程，中断服务例程做必要的数据收集（更新），之后唤醒高优先级任务。这个高优先级任务在中断服务例程结束后立即执行，它可能是其它任务也可能是中断处理任务，如果是中断处理任务，那么就可以根据中断服务例程中收集的数据做相应处理。

       configMAX\_SYSCALL\_INTERRUPT\_PRIORITY接口有着更深一层的意义：在优先级介于RTOS内核中断优先级（等于configKERNEL\_INTERRUPT\_PRIORITY）和configMAX\_SYSCALL\_INTERRUPT\_PRIORITY之间的中断允许全嵌套中断模式并允许调用API函数。大于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的中断优先级绝不会因为执行RTOS内核而延时。

       运行在大于configMAX\_SYSCALL\_INTERRUPT\_PRIORITY的优先级中断是不会被RTOS内核所屏蔽的，因此也不受RTOS内核功能影响。这主要用于非常高的实时需求中。比如执行电机转向。但是，这类中断的中断服务例程中绝不可以调用FreeRTOS的API函数。

      为了使用这个方案，应用程序要必须符合以下规则：调用FreeRTOS API函数的任何中断，都必须和RTOS内核处于同一优先级（由宏configKERNEL\_INTERRUPT\_PRIORITY设置），或者小于等于宏configMAX\_SYSCALL\_INTERRUPT\_PRIORITY定义的优先级。

37.configASSERT

       断言，调试时可以检查传入的参数是否合法。FreeRTOS内核代码的关键点都会调用configASSERT( x )函数，如果参数x为0，则会抛出一个错误。这个错误很可能是传递给FreeRTOS API函数的无效参数引起的。定义configASSERT()有助于调试时发现错误，但是，定义configASSERT()也会增大应用程序代码量，增大运行时间。推荐在开发阶段使用这个断言宏。

       举一个例子，我们想把非法参数所在的文件名和代码行数打印出来，可以先定义一个函数vAssertCalled，该函数有两个参数，分别接收触发configASSERT宏的文件名和该宏所在行，然后通过显示屏或者串口输出。代码如下：

#define configASSERT( ( x ) ) if( ( x ) == 0 )vAssertCalled( \_\_FILE\_\_, \_\_LINE\_\_ )

       这里\_\_FILE\_\_和\_\_LINE\_\_是大多数编译器预定义的宏，分别表示代码所在的文件名（字符串格式）和行数（整形）。

       这个例子虽然看起来很简单，但由于要把整形\_\_LINE\_\_转换成字符串再显示，在效率和实现上，都不能让人满意。我们可以使用C标准库assert的实现方法，这样函数vAssertCalled只需要接收一个字符串形式的参数（推荐仔细研读下面的代码并理解其中的技巧）：

#define STR(x) VAL(x)

#define VAL(x) #x

#define configASSERT(x) ((x)?(void) 0 :xAssertCalld(\_\_FILE\_\_ ":" STR(\_\_LINE\_\_) " " #x"\n"))

       这里稍微讲解一下，由于内置宏\_\_LINE\_\_是整数型的而不是字符串型，把它转化成字符串需要一个额外的处理层。宏STR和和宏VAL正是用来辅助完成这个转化。宏STR用来把整形行号替换掉\_\_LINE\_\_，宏VAL用来把这个整形行号字符串化。忽略宏STR和VAL中的任何一个，只能得到字符串”\_\_LINE\_\_”，这不是我们想要的。

       这里使用三目运算符’?:’来代替参数判断if语句，这样可以接受任何参数或表达式，代码也更紧凑，更重要的是代码优化度更高，因为如果参数恒为真，在编译阶段就可以去掉不必要的输出语句。

**38.INCLUDE Parameters:**

       以“INCLUDE”起始的宏允许用户不编译那些应用程序不需要的实时内核组件（函数），这可以确保在你的嵌入式系统中RTOS占用最少的ROM和RAM。

每个宏以这样的形式出现：

       INCLUDE\_FunctionName

      在这里FunctionName表示一个你可以控制是否编译的API函数。如果你想使用该函数，就将这个宏设置成1，如果不想使用，就将这个宏设置成0。比如，对于API函数vTaskDelete()：

#define INCLUDE\_vTaskDelete 1

       表示希望使用vTaskDelete()，允许编译器编译该函数

#define INCLUDE\_vTaskDelete 0

       表示禁止编译器编译该函数。